Space Transportation System
Command, Control and Monitor Application Software
Programming Standards

Contract NAS10-10900

Kennedy Space Center, Florida

1. INTRODUCTION

Good coding style should encourage consistent program layout, improve maintainability and portability and reduce errors. This standard provides rules and recommendations that will aid in reaching this consistency across the software development organizations. Of necessity, these standards cannot cover all situations. Experience and informed judgment count for as much as the standards. Ultimately, the goal of these standards is to increase portability, reduce maintenance and above all improve clarity.

1.1 PURPOSE

The purpose of this standard is to define one style of programming in C/C++. The standards and recommendations presented here are not final, but should serve as a basis for continued software development in these languages. This collection of standards should be viewed as a dynamic document; suggestions for improvement are encouraged.

Programs that are developed according to these standards and recommendations should be correct and easy to maintain. In order to attain these goals, the programs should:

a. Have a consistent style
b. Be easy to read and understand
c. Be free of common types of errors
d. Be maintainable by different programmers
e. Be portable to other architectures/platforms

1.2 SCOPE

This standard applies to all Application Software developed or maintained for use in the Space Transportation System. The standard covers programming style including naming conventions, program layout, syntax style and class (C++) design considerations. Questions of program design or architecture are beyond the scope of this document.

As a matter of convenience and efficiency, this document contains both rules and recommendations on software programming practices. Sections which are designated as rules must be adhered to without exception. Recommendations are provided as a guide for developing all software in as common a style as possible without imposing unnecessary restrictions. As such, adherence to the recommendations is suggested, but is not mandatory. However, deviation from the recommendations should only be done with good justification.

1.3 AUTHORITY

This document is internally controlled by the CLCS Project Manager or the appointed representative associated with the implementing department. All modifications to the document require this level of approval.

1.3.1 Applicability

This standard applies to all Command, Control and Monitor Application Software developed for the CLCS.

1.3.2 Standard Modification

A form for requesting new rules or changes to the rules of this standard has been included as an appendix to this document. All requests must be submitted to the standard responsible organizational representative (ROR) for evaluation.

1.4 REFERENCE RESOURCES

The following references were used in compiling the rules and recommendations of this standard:

Software and Automation Systems Branch C++ Programming Style, Version 1.0
Automation Systems Branch (Code 522), Goddard Space Flight Center, July 1992.

Programming in C++, Rules and Recommendations
Ellemtel Telecommunication Systems Laboratories, Alvsjo, Sweden, 1992.

Recommended C Style and Coding Standards
Bell Labs, Zoology Computer Systems, University of Toronto, CS University of Washington, 1989.

Writing Solid Code
Steve Maguire, Microsoft Press, Redomond, Washington, 1993.

C++ Programming Style
Tony Cargill, Addison-Wesley Publishing, Reading, Massachusetts, 1992.

2. EXECUTABLE PROGRAM NAMING CONVENTIONS

This section of the CLCS CCM Application Software Standard defines the system to be used for alphanumerically identifying all CCM application software used to monitor and control the STS Space Shuttle and related Ground Support Equipment (GSE) from the CLCS.

2.1 NAME CONTROL

The KSC NASA and contractor organizations responsible for the development of CCM application software shall also be responsible for the tracking of all program names. A register of all assigned names shall be maintained and no program name shall be reused once it has been issued. The mechanism for assigning program names shall be at the discretion of the responsible Engineering groups.

2.2 NAMING CONVENTION

RULE: All CCM Application Software program names shall be limited to a maximum of 32 characters in length. Two separate fields shall be used to define the owner and purpose of the program as defined in the following sections. The program name shall be formatted as follows:

SYS▪[Alpha/Numeric Identifier]

2.2.1 Responsible System Field

RULE: The first three characters of the name field shall identify the system associated with the program as defined by each responsible engineering group followed by a period delimiter.

APU	Auxiliary Power Unit
BHY	SRB Hydraulics (HPU/HYD)
CIN	CCS Integration
CMS	CCS Master
COM	Communications
DPM	DPSME
DPS	Data Processing System
ECL	ECLSS
ECS	Environmental Control System
EPD	Electrical Power Distribution
FSW	Flight Software
GLS	Ground Launch Sequencer
GNC	Guidance, Navigation & Control
HVC	CCS Heating, Ventilation & Air Conditioning
HWS	Hazardous Warning System
HYD	Orbiter Hydraulics
INS	Instrumentation
INT	Integration
LAC	Launch Accessories
LH2	Liquid Hydrogen System
LO2	Liquid Oxygen System
MEQ	Mechanisms
MPS	Main Propulsion System
MST	Master
NAV	Navigation
OAA	Orbiter Access Arm
OMS	Orbiter Maneuvering System
PNU	CCS Pneumatics System
PRS	Power Reactant Storage & Distribution System
PLD	Payload
PVD	Purge, Vent & Drain
PWR	CCS 60Hz Power
RCS	Reactive Control System
RSS	Range Safety System
SSM	Space Shuttle Main Engine
WTR	CCS Water System

2.2.2 Alpha/Numeric Identifier Field

RULE: The remaining field provides a 28 character alpha/numeric field for unique identification of the program. This field may include but is not limited to a method for identifying different subsystems within a system and a functional description of the program. Valid characters for this field shall include the upper and lowercase alpha characters, numerals 0 through 9, dash, underscore, ampersand, and period.

3. PROGRAMMING RULES AND RECOMMENDATIONS

3.1 FILE ORGANIZATION

3.1.1 Source Code Contents

RULES: All source code files shall have the following sections in the order specified. If a section does not apply, then it is not necessary to include it. Each applicable section shall be preceded by a section title comment.

1. All program source code files shall include an introductory comment that provides a description of the module and a revision history that includes the ending revision number, WAD number(s) and description of the change(s), the name and organization of the programmer making the change(s), and the date of the change(s). An example of the required introductory comment is provided in Appendix A.

   RECOMMENDATION: Introductory comments should include descriptions of any interprocess communications techniques and their uses (e.g. shared memory, streams, etc.) as well as anything unique about how the module is designed/coded.

   RULE: All files shall include the following copyright statement in the introductory comment:

   Copyright (year) National Aeronautics and Space Administration
   All Rights Reserved

   where year represents the year the program baseline revision was created.

2. System include files (e.g. <iostream.h>)

3. Application specific include files (e.g. “VBaseApplication.h”)

4. External functions

5. External variables

6. Constants

7. Macros

8. Class Declarations

9. Non-member functions

3.1.2 File Size

RECOMMENDATION: File size should be limited to approximately 1000 lines of source code. Although there is no real maximum length for source files, those with more than 1000 lines are cumbersome to deal with. Editors may not have enough temporary space to edit the file and compilations will go slower.

3.1.3 Line Length

RECOMMENDATION: Line lengths should be limited to 79 characters. Lines longer than 79 characters are not handled well by all terminals and should be avoided if possible. Excessively long lines which result from deep indentation are often a symptom of poorly organized code.

Source code lines should not wrap unless absolutely necessary, as in the definition of a long literal. This can lead to confusion about program structure.

3.1.4 Include File Contents

RULE: An application include file shall contain all other necessary include files to support the classed referenced in the file. It is more maintainable to have all supporting files automatically included when a class include file is used than requiring the user to remember to include those support files.

3.1.5 Multiple Inclusion Prevention

RULE: Every include file shall contain a mechanism that prevents multiple inclusions of the file. The easiest way to avoid multiple includes of a file is by using the ***#ifndef/#define*** block at the beginning of the file and ***#endif*** at the end of the file.

3.2 APPLICATION FILE NAMING CONVENTIONS

The purpose of these conventions is to provide a uniform interpretation of file names. One reason for this is that it is easier to develop and/or use tools which base their behavior on file name extensions.

3.2.1 Required Files

RULE: Each class shall be represented by two source code files: the interface definition (or header) file and the implementation file.

Approved Exception: Two closely interrelated classes may be combined into a single pair of header and implementation files.

3.2.2 Application Filenames

RECOMMENDATION: Always give a file a name that is unique in as large a context as possible. Since class names must generally be unique within a large context, it is appropriate to utilize this characteristic when naming its implementation and header files. This convention makes it easy to locate a class definition using a file-based tool.

RULE: A header file shall have the same name as its associated implementation file (e.g. BaseClass.C and BaseClass.h)

RULE: Interface definition (header) files shall not have filenames that have a system complement (e.g. “math.h” and <math.h>). The statement *#include “math.h”* will include the standard library math include file if the intended one is not found in the current directory.

3.2.3 Application Source Code Filename Extensions

RULE: The extensions listed in the following table shall be used for all source code files.

Extension	File Type
.C	C++ implementation file
.c	C implementation file
.h	All application header files

3.3 LIBRARY FILE NAMING CONVENTIONS

3.3.1.1 Library Filenames

RECOMMENDATION: Library filenames should be unique in as large a context as possible. The name should reflect the function of the library, as well as indicate that it is a library.

3.4 SOURCE CODE COMMENTS

RECOMMENDATIONS: It is essential to document source code to capture the thought processes behind the software development. It may also be helpful to include a requirements tracking cross-reference. This should be compact and easy to find. By properly choosing names for variables, functions and classes and by properly structuring the code, there is less need for lengthy comments within the code.

Note that comments in include files are meant for the users of classes, while comments in implementation files are meant for those who maintain the code. A tactical comment describes what a single line of code is intended to do and is placed, if possible, at the end of the line. Unfortunately, too many tactical comments can render the code unreadable.

3.4.1 Comment Maintenance

RULE: Comments shall be maintained with the same attention as the associated code. Comments that disagree with the code are of negative value. The importance of keeping the comments in synchronization with the code cannot be over emphasized.

3.4.2 Comment Syntax

RECOMMENDATION: Use the "//" syntax for all comments (C++ compilers only). If the characters "//" are used consistently for writing comments, then the combination of "/* */" may be used to make comments out of entire sections of code during development and debug phases.

3.4.3 Strategic Comments

A strategic comment describes what a function or section of code is intended to do and is normally placed before the code.

RULE: All strategic comments shall appear before the section of code they document. Putting a comment at the top of a 3 - 10 line section explaining the purpose of the code is often more useful than a comment on each line describing the micrologic.

RULE: Every function shall contain a strategic comment before the function declaration explaining its purpose and anything special about the function. It is also helpful to add a comment for the parameter list which describes the purpose of each parameter.

3.5 FUNCTION/VARIABLE NAMING CONVENTIONS

RECOMMENDATION: When naming a class, function or variable, the names should be as clear as possible. The goal should be to make the user's interface conceptually transparent. The code will be more understandable and readable when names are closely related with their associated purpose.

3.5.1 Naming Conventions

RULES: The following conventions shall be followed for all class, function and variable names. Underscores shall not be used in any user application name. All names shall be mixed case with the initial letter of each word capitalized (e.g. BaseClass), with further distinction by Type as defined with the exception of constants.

1. Virtual base classes shall begin with a capital 'V' with the first letter of each word capitalized thereafter(e.g. VBaseClass).

2. Function names shall be mixed case with the initial letter of each word capitalized (e.g. OpenValve).

3. Class member attributes shall begin with a lower-case 'm' with the first letter of each word capitalized thereafter (e.g. mMemberObject).

4. Global variables (which should be avoided if at all possible) shall begin with a lower-case 'g' with the first letter of each word capitalized thereafter (e.g. gGlobalVariable).

5. Local variables shall begin with the first word in lower-case with the first letter of each word capitalized thereafter (e.g. internalVariable).

6. Constants (***const*** and ***enum***) and macros shall be named in all upper-case letters (e.g. HILIM).

3.5.2 Naming Recommendations

The following conventions should be followed when naming classes, functions and variables. Adherence to these suggestions will greatly enhance the maintainability of the software.

1. ***RECOMMENDATION:*** Choose names that suggest the usage. One rule of thumb is that a name which cannot be pronounced is a bad name. A long name is normally better than a short, cryptic one, but the truncation problem must be taken into consideration. Abbreviations can always be misunderstood. Global variables, functions and constants should have long enough names to avoid name conflict, but not unreasonably long.

2. ***RULE:*** Names that differ only by the use of upper-case and lower-case letters shall not be used.

3. ***RECOMMENDATION:*** Names should not include abbreviations that are not generally accepted.

3.6 OPTIMIZATION

3.6.1 Condition Assertion

RECOMMENDATION: The ***assert*** macro (contained in ***<assert.h>***) should be used whenever possible to assist in the detection and isolation of errors. ***assert*** is a debug-only macro that aborts execution if its argument is false. ***assert*** should not disturb memory or initialize data that would otherwise be uninitialized or cause any other side effects.

3.6.2 Performance Enhancements

RECOMMENDATIONS: Optimize code only if it has a known performance problem. Various industry tests have demonstrated that programmers generally spend a lot of time optimizing code that is seldom or never executed. If a program's performance is not acceptable, determine the exact nature of the problem before starting optimization activities.

Performance measurement development tools (e.g. Pure Software, Quantify, CenterLine TestCenter) should be used to provide performance measurements that can be used to improve performance and isolate run-time errors.

3.7 PORTABILITY

C/C++ code is not inherently portable. Thought and effort are required to make it so. As a general practice, all other things being equal, portable code is better than non-portable code. This section provides the rules and recommendations for portability standardization.

3.7.1 Main Routine Form

RULE: The heading for main shall be defined as either:

int main (void)
int main (int argc, char *argv[ ])

These are the forms explicitly sanctioned by the ANSI-C standard. Other common forms, such as ***void main (void)*** only work on some platforms.

3.7.2 Program Exit Codes

RULE: The constants ***EXIT_SUCCESS*** and ***EXIT_FAILURE*** shall be used as program exit codes. Calling ***exit(0)*** is portable, but ***exit(1)*** is not. A return statement in main effectively calls exit, so main should normally return ***EXIT_SUCCESS***. These constants are defined in ***<stdlib.h>*.**

3.7.3 Object Size Determination

RULE: The generic type ***size_t*** (defined in several standard headers) shall be used as the type of an object that holds the size of another object. ***size_t*** is always defined as an unsigned integer type, which varies from platform to platform. ***unsigned long int*** could be used in place of ***size_t***, but it may be wasteful on some architectures. ***size_t*** is always the right size (e.g. ***size_t strlen (const char*)***).

3.7.4 Pointer Difference Calculation

RULE: When the difference between two pointers is required, the value shall be stored in an object of type ***ptrdiff_t*** (defined in ***<stddef.h>***). Each implementation defines ***ptrdiff_t*** as a signed integer type of the appropriate size for the target architecture.

3.7.5 User Unique Data

RECOMMENDATION: The generic types of ***void*** and ***void(*)()*** should be used for user unique data objects and functions. The ***void*** type is guaranteed to have enough bits of precision to hold a pointer to any data object. The ***void(*)()*** type is guaranteed to be able to hold a pointer to any function. Be sure to cast pointers back to the correct type before using them.

3.7.6 Long Data Type

RECOMMENDATION: Be very careful defining data structures using the ***long*** data type. If a stream moves data between platforms with different base register sizes (e.g. HP uses 32-bit, DEC Alpha uses 64-bit), the effective size of the ***long*** will cause data structures to be misaligned.

3.7.7 System Calls

RULE: System calls shall not be used that are platform dependent.

3.7.8 External File References

RECOMMENDATION: When referencing external files and/or programs, specify the path using either a configuration file or environment variable. Avoid hard-coded pathnames. The use of hard-coded pathnames will cause the re-code of modules when the file structure changes. Also, do not assume a user's ***PATH*** variable includes the directory for what should be a common file. Always specify the full pathname for all files and/or programs used.

3.7.9 Temporary Files

RECOMMENDATIONS: When using temporary files, use the UNIX function ***tempnam*** to ensure unique filenames.

Use a directory other than ***/tmp*** for temporary file storage. On some systems, the /tmp directory usage is restricted by the System Administrator. Using this directory to store temporary files in this circumstance will cause the read/write to fail.

4. CODING STANDARDS

4.1 GENERAL CODING

4.1.1 Braces Placement

RULE: Braces "{ }" which enclose a block of code shall be placed in the same column on separate lines directly before and after the block. The placement of braces seems to have been the subject of the greatest debate concerning code appearance. For consistency across the project, this style shall be used:

```
if (condition == TRUE)
{
        fTrueFunction1( );
        fTrueFunction2( );
}
else
{
        fFalseFunction1( );
        fFalseFunction2( );
}
```

4.1.2 Operator Usage

RULE: Spaces around the "**.**" and "***->***" operators shall not be used. Code is more readable if spaces are not used in these instances.

4.1.3 Flow Control Statements

4.1.3.1 ***RECOMMENDATION:*** The flow control statements ***while, for*** and ***do*** should be followed by a block of code, even if it is empty. At times, everything that is to be done in a loop can easily be written on one line in the loop statement. It may then be tempting to conclude the statement with a semicolon at the end of the line. This may lead to misreading of the code since the semicolon may be missed. It seems to be better to place an empty block of code after the statement to make it completely clear what the code is doing.

4.1.3.2 ***RULE:*** No more than one statement shall be included per line of source code.

4.1.3.3 Error handling needs to be addressed here.

4.1.3.4 ***RULE:*** The code following a case label shall always be terminated by a ***break/return*** statement. When several case labels are followed by the same block of code, only one ***break/return*** statement is required. If the code which follows a ***case*** label is not terminated by a ***break/return***, the execution continues after the next ***case*** label. This means that poorly tested code can be erroneous and still seem to work.

4.1.3.5 ***RULE:*** A ***switch*** statement shall always contain a ***default*** branch which handles unexpected cases.

4.1.3.6 ***RECOMMENDATION:*** Use the ***goto*** syntax with caution and deliberation. ***goto*** breaks the control flow and can lead to code that is difficult to comprehend and maintain. For extremely time critical applications, ***goto*** may be permitted. Every such usage must be carefully motivated, and should be explained in a comment.

4.1.3.7 ***RECOMMENDATION:*** Use parenthesis to clarify the order of evaluation for operations in expressions. If the operator precedence is not obvious or the expression is complicated or long, use parenthesis to make it more readable. Even if the parenthesis are not technically required, make the order of evaluation obvious.

4.1.3.8 ***RECOMMENDATION:*** Check the fault codes which may be received from library functions even if the function seems foolproof. Two important characteristics of a robust system are that all faults are reported and, if the fault is so serious that continued execution is not possible, the process is terminated. In this way the propagation of faults through the system is avoided. In achieving this, it is important to always test fault codes from library functions (e.g. opening/closing files, allocation of memory for data). One test too many is better than one test too few. Application specific functions should preferably not return fault codes but should instead take advantage of exception handling features.

4.1.4 Memory Allocation

4.1.4.1 ***RULE:*** ***malloc, realloc*** or ***free*** shall not be used for memory allocation in C++ only. In C, ***malloc, realloc*** and ***free*** are used to allocate memory dynamically on the heap. This may lead to conflicts with the use of the ***new*** and ***delete*** operators in C++. It is dangerous to:

1. Invoke ***delete*** for a pointer obtained via ***malloc/realloc***.
2. Invoke ***malloc/realloc*** for objects having constructors.
3. Invoke ***free*** for anything allocated using ***new***.

4.1.4.2 ***RULE:*** Empty brackets "**[ ]**" shall always be provided for delete when deallocating arrays (C++ only). If an array 'p' having a type 'T' is allocated, it is important to invoke ***delete*** in the correct way:

1. (WRONG) ***delete p*** results in the destructor being invoked only for the first object of type T.
2. (WRONG) ***delete [m] p*** where 'm' is an integer which might be greater than the number of objects allocated earlier, the destructor for 'T' will be invoked for memory that does not represent objects of type T.
3. (CORRECT) ***delete [] p*** is the correct way since the destructor will then be invoked only for those objects which were allocated earlier.

4.1.4.3 ***RULE:*** Memory that is allocated shall be deallocated when it is no longer needed. Do not allocate memory and expect that someone else will deallocate it later. For instance, a function can allocate memory for an object which is then returned to the user as the return value for the function. There is no guarantee that the user will remember to deallocate the memory, and the interface with the function then becomes considerably more complex.

4.1.4.4 ***RULE:*** A pointer that points to deallocated memory shall always be assigned to a new value. Pointers that point to deallocated memory should either be set to 0 or be given a new value to prevent access to released memory. This can be a very difficult problem to solve when there are several pointers which point to the same memory since C++ has no garbage collection.

4.1.4.5 ***RECOMMENDATION:*** Avoid frequently allocating and deallocating memory (C only) which may cause memory fragmentation.

4.1.5 Standard Error Handling

Standard Error Handling needs to be addressed here.

4.2 CLASS CODING (C++ Only)

4.2.1 Class Definitions

4.2.1.1 ***RULE:*** The public, protected and private sections of a class definition shall be declared in that order. Both member attributes and member functions shall be declared in the same appropriate section. By placing the public section first, everything that is of interest to a user is gathered at the beginning of the class definition. The protected section may be of interest to designers when considering inheriting from the class. The private section contains details that should have the least general interest.

4.2.1.2 ***RECOMMENDATION:*** Make classes as simple as possible. Give each class a clear purpose. If classes grow too complicated, make more classes: break complex classes into simpler ones.

4.2.1.3 ***RECOMMENDATION:*** Friends of a class should be used to provide additional functions that are best kept outside of the class. A friend is a non-member of a class that has access to the non-public members of the class. Friends offer an orderly way of getting around data encapsulation for a class. Friends are good if used properly, but the use of many friends can indicate that the modularity of the system is poor.

4.2.1.4 ***RULE:*** Multiple inheritance shall not be used. It is for getting out of bad situations, especially repairing interfaces where control over the broken class belongs to someone else. The complexities of multiple inheritance override its usefulness in most situations.

4.2.1.5 ***RECOMMENDATION:*** Polymorphism needs to be addressed here. No downcasting should be used.

4.2.2 Required Class Functions

4.2.2.1 ***RULE:*** All class definitions shall have the constructor, destructor and assignment operator (***operator=***) defined. Don't let the compiler create these functions. Class designers should always say exactly what the class should do and keep the class entirely under their control. If a copy

constructor or assignment operator is not desired, declare it private. Remember, if any constructor is specified, it prevents the default constructor from being synthesized.

4.2.2.2 ***RULE:*** A class which uses ***new*** to allocate instances managed by the class or has pointers shall define a ***copy*** constructor. A ***copy*** constructor is recommended to avoid surprises when an object is initialized using an object of the same type. If an object manages the allocation and deallocation of objects on the heap, only the value of the pointer will be copied. This can lead to two invocations of the destructor for the same object, probably resulting in a run-time error.

4.2.2.3 ***RULE:*** All classes which are used as base classes and which have virtual functions shall define a ***virtual*** destructor. If a class having virtual functions but without virtual destructors is used as a base class, there may be a surprise if pointers to the class are used. If such a pointer is assigned to an instance of a derived class and if ***delete*** is then used on the pointer, only the base class' destructor will be invoked. If the program depends on the derived class' destructor being invoked, it will fail.

4.2.2.4 ***RULE:*** If a public base class does not have a virtual destructor, no derived class nor members of a derived class should have a destructor. If a derived class or member of a derived class defines a destructor and the base class destructor remains non-virtual, memory leaks or other abnormalities can occur.

4.2.3 Member Function Rules and Recommendations

4.2.3.1 ***RULE:*** Member functions shall only be prototyped within a class definition. A member function that is defined within a class definition automatically becomes an in-line function. Class definitions are less compact and more difficult to read when they include definitions of member functions. It is easier for an in-line member function to become an ordinary member function if its definition is placed outside of the class definition.

4.2.3.2 ***RULE:*** A member function that does not affect the state of an object shall be declared ***const***. Member functions declared as ***const*** may not modify member data and are the only functions which may be invoked on a ***const*** object. A ***const*** declaration is excellent insurance that objects will not be modified when they should not be. ***const*** member functions may never be invoked as an "lvalue" (a location value where a value may be stored).

4.2.3.3 ***RULE:*** A public member function shall never return a non-***const*** reference or pointer to member data. By allowing a user direct access to the private member data of an object, this data may be changed in ways not intended by the class designer.

4.2.3.4 ***RULE:*** Do only what is minimally necessary in constructors. Not only does this produce a lower overhead for constructor calls, but the constructors are also less likely to throw exceptions or cause problems. Use initialization routines if necessary to initialize class attributes.

4.2.3.5 ***RULE:*** All functions shall have a prototype definition. This practice helps eliminate function calling errors that might otherwise have been avoidable. This purpose can be strengthened by

making the argument types more accurate (e.g. ***unsigned ch*** instead of ***int***). The drawback to this is that it may be necessary to cast arguments to silence noncritical type mismatch warnings.

4.2.3.6 ***RULE:*** The names of formal arguments to functions shall be specified and are to be the same in both the function declaration and in the function definition. The names of formal arguments may be specified in both the function declaration and definition in C++, even if these are ignored by the compiler in the declaration. Providing for function arguments is part of the function documentation. The name of the arguments may clarify how they are used, reducing the need to include comments for documenting that purpose.

4.2.3.7 ***RULE:*** The return type of a function shall always be provided explicitly. If no return type is explicitly provided, it is, by default, an ***int***. To improve the readability and clarity of the code, function return types must be specified.

4.2.3.8 ***RULE:*** A public function shall never return a reference or pointer to a local variable. If a function returns a reference or pointer to a local variable, the memory to which it refers will already have been deallocated when the reference or pointer is used. The compiler may or may not give a warning of this.

4.2.3.9 ***RULE:*** When two operators are opposite (such as "==" and "!="), both shall be defined even if only one is necessary.

4.2.3.10 ***RECOMMENDATION:*** When declaring functions, the leading parenthesis and the first argument (if any) should be written on the same line as the function name. If space permits, other arguments and the closing parenthesis may also be on the same line. Otherwise, each additional argument should be written on a separate line (with the closing parenthesis directly after the last argument).

4.2.3.11 ***RECOMMENDATION:*** Avoid functions with many arguments. Functions having long lists of arguments look complicated, are difficult to read, and can indicate poor design. In addition, they are difficult to read and to maintain.

4.2.3.12 ***RECOMMENDATION:*** Pass arguments by ***const*** reference as the first choice. As long as the object being passed in does not need to be modified, this practice is best because it has the simplicity of pass-by-value syntax but does not require expensive constructions and destructions to create a local object.

4.2.3.13 ***??? RECOMMENDATION:*** If a function stores a pointer to an object which is accessed via an argument, the argument should have the type pointer. Use reference arguments in other cases. By using references instead of pointers as function arguments, code can be made more readable, especially within the function. A disadvantage is that it is not easy to see which functions change the values of their arguments.

4.2.3.14 ***RECOMMENDATION:*** Watch for overloading. A function should not conditionally execute code based on the value of an argument (default or not). In this case, two or more overloaded functions should be created.

4.2.3.15 ***RECOMMENDATION:*** Use overloaded function names instead of different function names to distinguish between functions that perform the same operations on different data types. Remember, C++ does not permit overloading on the basis of the return type.

4.2.3.16 ***RECOMMENDATION:*** Consider default arguments as an alternative to function overloading. In general, replacing function overloading by a default argument makes a program easier to maintain because there is only one copy of the function body (e.g. function A (int a, char b, x=1)).

4.2.3.17 ***RECOMMENDATION:*** When overloading functions, all variations should be used for the same purpose. Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for a different purpose), they can, however, cause considerable confusion.

4.2.3.18 ***RECOMMENDATION:*** Avoid long and complex functions. If a function is too long it can be difficult to comprehend. Generally, it can be said that a function should be no longer than two pages since that is about how much that can be comprehended at one time.

If an error situation is discovered at the end of an extremely long function, it may be difficult for the function to clean-up after itself and to “undo” as much as possible before reporting the error to the calling function. By using short functions, such an error can be more exactly localized.

Complex functions are also much more difficult to test.

4.2.4 In-Line Member Function Rules and Recommendations

4.2.4.1 ***RULE:*** In-line functions shall not be defined in the interface definition (header) file for the class.

4.2.4.2 ***RECOMMENDATION:*** Access and forwarding functions should be in-line member functions. This will improve the performance of the class.

4.2.4.3 ***RULE:*** Constructors and destructors shall not be defined as in-line functions. A constructor always invokes the constructors of its base classes and member data before executing its own code. This cascading of in-line constructors may be too complex for some compilers to handle efficiently.

4.2.4.4 ***RULE:*** In-line functions and parameterized types shall be used instead of preprocessor macros. This allows more parameter checking to be performed at compilation.

4.2.4.5 ***RECOMMENDATION:*** Use of in-line functions in small programs can help performance. Extensive use of in-line functions in large projects can actually hurt performance by enlarging code, causing paging problems and forcing many recompilations.

4.2.5 Member Attribute, Variables and Constants

4.2.5.1 ***RULE:*** In variable declarations, the pointer qualifier (*) shall be with the variable name rather than with the type. Declare variables as ***char *s, *t, *u;*** rather than ***char* s, t, u;*** In the latter instance, only ***s*** is declared as a character pointer.

4.2.5.2 ***RULE:*** Public and protected member data shall not be specified in a class definition. A public variable represents a violation of one of the basic principles of object oriented programming, namely, data encapsulation. Access functions should be used to return values of private member data. This avoids the possibility of an arbitrary function changing the public data value which may lead to errors that are difficult to locate.

4.2.5.3 ***RULE:*** Symbolic values shall be defined instead of numeric values in code. Numerical values in code can be the cause of difficult problems if and when it becomes necessary to change a value. A large amount of code can be dependent on such a value never changing and the value can be used at a number of places in the code, leading to difficulty in locating all instances of them.

4.2.5.4 ***RULE:*** Constants shall be defined using ***const*** or ***enum*** instead of ***#define.*** The preprocessor performs a textual substitution for macros in the source code which is then compiled. This can lead to a number of negative consequences. Names declared with ***#define*** are untyped and unrestricted in scope. IN contrast, names declared with ***const*** are typed and follow C++ scope rules.

4.2.5.5 ***RECOMMENDATION:*** Variables should be declared with the smallest possible scope. A variable ought to be declared with the smallest scope possible to improve the readability of the code and so variables are not unnecessarily allocated.

4.2.5.6 ***RULE:*** Every variable that is declared shall be given a value before it is used. A variable must be initialized before it is used. Normally the compiler gives a warning if a variable is undefined. Instances of a class are usually initialized even if no arguments are provided in the declaration (the empty constructor is invoked). By initializing all variables before they are used, the code is made more efficient since no temporary objects are created for the initialization. For objects having large amounts of data, this can result in significantly faster code. To declare a variable that has been initialized in another file, the keyword ***extern*** is always used.

4.2.5.7 ***RULE:*** Pointers shall not be assigned a value of ***NULL*** and shall not be compared to ***NULL***. A value of ***0*** (zero) shall be used instead. According to the ANSI-C standard (and the pending ANSI-C++ standard), ***NULL*** is defined as ***(void *)0*** or as ***0***. This may lead to errors unless an explicit type conversion is supplied.

4.2.5.8 ***RECOMMENDATION:*** Use ***unsigned*** for variables which cannot reasonably have negative values.

4.3 AUTO GENERATED CODE

Code generated by commercial tools needs to be addressed here.

5. MAKEFILES

The following methods include support for using appropriate compiler flags which ensure portability. It is strongly recommended that the application software standard Makefile template is used.

5.1 TARGETS

5.1.1 Clean Target

RULE: All Makefiles shall include a ***clean*** target. The ***clean*** target restores the directory to the state it would be in if the source had just been checked-out from the source code library. The Makefile invokes the ***clean*** target for Makefiles in directories below it.

5.1.2 All Target

RULE: All Makefiles shall include an ***all*** target. The ***all*** target makes everything in that directory. The Makefile invokes the ***all*** target for Makefiles in directories below it.

5.1.2.1 Default Target

RULE: All Makefiles shall include a ***default*** target. The ***default*** target must be the uppermost target in the Makefile. Its sole function is to remind the user to invoke the Makefile through one of the appropriate targets.

5.1.2.2 Makedepend Target

RECOMMENDATION: All Makefiles should include a ***makedepend*** target to assist in the setting of dependencies. The ***makedepend*** target invokes the ***makedepend*** utility. The Makefile dependencies are then automatically generated and added to the Makefile.

5.2 TAGS

RECOMMENDATION: ***Tags*** should be included in the Makefile when using the EMACS editor. The ***Tags*** function results in EMACS setting flags to identify function names, variable names, etc. When you want to locate the definition of the function or variable, you simply click on the name and EMACS searches for and displays the location of the definition.

5.3 MACROS

RULE: The following macros shall be defined in all Makefiles:

5.3.1 The CC Macro

RULE: The CC Macro is set to the requested compiler (cc, acc or gcc) along with a switch indicating the optimize or debug level for the compiler. A switch for ANSI-C and C++ is also available.

5.3.2 The LLIBS Macro

RULE: The LLIBS Macro gives the link library for C Code. All libraries required for proper linking shall be included in this macro definition.

6. USER INTERFACE STANDARDS

This section defines the programming standards that are applicable to the user display monitor interfaces and control interfaces.

6.1 *INTERFACE ATTRIBUTES*

6.1.1 Title

RULE: Each display monitor shall have a title that identifies the system/subsystem being monitored and the function of the monitor. The display program name shall be displayed as the display window frame title.

6.1.2 Program Activity Indicator

RULE: Each display monitor shall have a location in the upper right corner of the skeleton that shall be used to indicate activity of the associated software. This activity indication shall toggle between two distinct symbols and shall be periodically updated by the driving software (either on a cyclic or timed callback event basis).

6.1.3 Popup Windows/Menus

RULE: Popup Menus associated with cursor control targets shall have the target identifier as the menu title, shall not be moveable, and shall contain separator bars between external control menu items. Popup window displays shall follow the standards for displays.

6.1.4 Data Validity

Validity status of all data sources is provided through a system software user interface window (e.g. FEP, HIM, OI/GPC format).

6.1.5 Window Attributes

RULE: Display size and resizing options shall be defined by the responsible systems and specified in the display requirements. Message windows shall be scrollable when the data size exceeds the display window size.

6.2 *COLOR USAGE*

RULE: The colors identified in this section are reserved for the data types they represent. Use of these colors for display shall adhere to the following rules. Use of all other colors is at the discretion of the Responsible Engineering Groups. The usage shall be documented in the system display requirements and shall be consistent throughout the system's application software set.

6.2.1 Normal Text Data

RULE: Normal text data is defined as data that is valid and is within nominal limits. Textual data that is normal shall be displayed in GREEN.

6.2.2 Invalid Text Data

RULE: Invalid text data is defined as data that is either no longer being updated by CLCS, or is not supported by the current OI/GPC format. Textual data that is invalid shall be displayed in WHITE.

6.2.3 Discrepant/Cautionary Text Data

RULE: Discrepant or cautionary text data is defined as data that is valid but is outside of the normally accepted value or limits. Textual data that is discrepant or cautionary shall be displayed in YELLOW.

6.2.4 Error/Urgent Text Data

RULE: Error or urgent text data is defined as data that is valid but is outside of the normally accepted value or limits and represents an error or urgent condition that requires immediate resolution. Textual data that is in error or urgent shall be displayed in RED.

6.2.5 Background Color

RULE: All displays shall be built using one of the following colors designated as background colors: TBD. (Note: the display team is investigating the colors to designate as background colors based on the reserved color list. This item will be completed after the study is complete.)

6.3 *SYMBOLS*

RULE: All displays shall be built from the standard set of display object symbols located in the symbol library. Each symbol shall be reserved to represent a specific object or type of objects (e.g. one symbol may represent a ball valve, clay valve and globe valve) and shall only be used for that purpose. All new symbols must be included in the symbol library prior to their use on a display.

6.3.1 Symbol Library

RULE: The symbol library shall contain all approved display object symbols used on application software displays.

6.3.2 Animation

RULE: Use of animation shall be defined by the system in the display requirements.
(Note: performance issues of animation are being examined by the display team and needs to be addressed further in this document as more is learned.)

6.4 *CONTROL INTERFACES*

This section describes the standards that are applicable to all control interfaces.

6.4.1 Cursor Control Points

RULE: Specific locations on a display are identified as targets for issuing commands to both internal and external receptors. When the cursor passes over a control point, the cursor symbol shall change to indicate that it is over an active target, and return to the normal symbol when it leaves the control point. The cursor entering an active control point shall not automatically cause any other action to occur.

6.4.1.1 *Display Control Point Identification*

RULE: All cursor control points or targets used to issue external stimuli shall have a text descriptor adjacent to the target symbol identifying the target's function. Each descriptor on a given display shall be unique.

6.4.1.2 *Emergency/Safing Control*

RECOMMENDATION: Emergency and safing control may be performed through a single step action which is uniquely identified on the display. These actions shall be defined by the systems in the display requirements.

6.4.1.3 *Normal Control*

RULE: All normal cursor control points shall have a popup menu as the interface associated when the target is armed. The popup menu shall have the target identifier as the menu title, shall not be moveable, and shall contain separator bars between externally controlled item selections. All control shall be initiated through this menu interface.

6.4.1.4 *Alternate Control*

RULE: If the primary mechanism for issuing commands and functions is via the mouse, a secondary keyboard method shall also be provided.

6.4.2 Mouse Control

The basic definitions for each mouse key are as follows:

6.4.2.1 *Mouse Key 1*

RULE: All mouse initiated cursor control points shall be initiated by the operator using mouse key 1. The keyboard equivalent of this is the Enter key.

6.4.2.2 *Other Mouse Keys*

RECOMMENDATION: Other mouse keys may be used for all other purposes except as stated above.

6.4.3 Programmable Function Key Interfaces

RULE: Programmable Function Keys shall be defined by the systems and usage shall be documented in the system design documents. These keys shall be consistent throughout the system's application software set.

6.4.4 Programmable Function Panel Interfaces

RULE: Programmable Function Panel Keys shall be defined by the systems and usage shall be documented in the system design documents. These keys shall be consistent throughout the system's application software set.

6.5 *MESSAGE NOTIFICATION*

RULE: All application software messages shall be identified with a date/time tag to milliseconds, and shall include the functional program source of the message. Messages written to the application software message window shall also be displayed in color as defined below.

6.5.1 Informational Messages

RULE: Informational messages shall be displayed in a color defined by the system and shall be defined in the display requirements. This color shall be consistent throughout the system's application software set.

6.5.2 Caution Messages

RULE: Caution messages shall follow the standard defined for cautionary text data and be displayed in YELLOW. All caution messages shall also be recorded for archival (SPA equivalent).

6.5.3 Error Messages

RULE: Error messages shall follow the standard defined for error text data and be displayed in RED. All error messages shall also be recorded for archival (SPA equivalent).

6.5.4 Prompts

RULE: All prompts shall be displayed to a prompt message window and be obvious that a user response is required. The prompt interface shall be defined in the application software architecture document as a user response window.

APPENDIX A COMMENT FORMATS

6.6 INTRODUCTOR COMMENTS

RULE: The following format shall be used for all source code introductory comments.

```
//*************************************************************************************************************
//**  Name:            Source file name                                                                      **
//**                                                                                                         **
//**  Description:     A brief description of the file's contents                                            **
//**                                                                                                         **
//**  Notes:           Brief programming notes describing any peculiar aspects of the code                   **
//**                   contained in the file.  Any detailed descriptions will appear in each                 **
//**                    individual function's comment header.                                                **
//**                                                                                                         **
//**  Warnings/Limitations:                                                                                  **
//**                   Brief description of any shortcomings of the code in this file.  Any                  **
//**                   detailed description will appear in each individual function's comment                **
//**                   header.                                                                               **
//**                                                                                                         **
//**  Revision History:                                                                                      **
//**  ---------------------                                                                                  **
//**   Ending Rev No       WAD number and description of change driver and of change                         **
//**                       implemented                                                                       **
//**                       Programmer and Organization                                                       **
//**                       Date                                                                              **
//**                                                                                                         **
//**  Copyright (base rev year)  National Aeronautics and Space Administration                               **
//**                                 All rights reserved.                                                    **
//*************************************************************************************************************
```

6.7 FUNCTION COMMENT HEADER

RULE: Each function within a source file shall have an associated comment header that provides the following information:

```
//*************************************************************************************************
//**  Name:         Function name                                                                 **
//**                                                                                              **
//**  Description:  A brief description of the function's purpose                                 **
//**                                                                                              **
//**  Arguments:    arg1 - description of arg1                                                    **
//**                arg2 - description of arg2                                                    **
//**                                                                                              **
//**  Returns:      Description of the value that is returned.                                    **
//**                                                                                              **
//**  Notes:        Detailed programming notes describing the function's design, the              **
//**                decision processes associated with the development.  These notes              **
//**                should not make an attempt to describe the functionality of the code.         **
//**                Functionality issues should be described with block comments or               **
//**                inferred from the code.                                                       **
//*************************************************************************************************
```

6.8 SECTION COMMENT HEADER

RULE: Each section of code shall have an associated comment header that provides the title of the section as follows:

```
//*************************************************************************************************
//**                                                                                              **
//**                                      SECTION TITLE                                           **
//**                                                                                              **
//*************************************************************************************************
```

APPENDIX B MAKEFILE TEMPLATES

6.9 SOURCE CODE MAKEFILE TEMPLATE

RECOMMENDATION: The following example is the template for the Makefile for source code:

TBD

6.10 SL-GMS DISPLAY MAKEFILE TEMPLATE

RECOMMENDATION: The following example is the template for the Makefile for SL-GMS generated displays:

TBD